Rcpp Attributes

J.J. Allaire Dirk Eddelbuettel Romain François

Rcpp version 0.10.0 as of November 13, 2012

Abstract

Rcpp attributes provide a high-level syntax for declaring C++ functions as callable from R and automatically generating the code required to invoke them. Attributes are intended to facilate both interactive use of C++ within R sessions as well as to support R package development. Attributes are built on top of **Rcpp** modules and their implementation is based on previous work in the **inline** package (Sklyar, Murdoch, Smith, Eddelbuettel, and François, 2012).

1 Introduction

Rcpp attributes are a new feature of **Rcpp** version 0.10.0 (Eddelbuettel and François, 2012, 2011) that provide infrastructure for seamless language bindings between R and C++. The motivation for attributes is several-fold:

1. Reducing the learning curve associated with using C++ and R together
2. Eliminating boilerplate conversion and marshaling code wherever possible
3. Seamless use of C++ within interactive R sessions
4. Unified syntax for interactive work and package development

The core concept is to add declarative attributes to C++ source files that provide the context required to automatically generate R bindings to C++ functions. Attributes and their supporting functions include:

- `Rcpp::export` attribute to export a C++ function to R
- `sourceCpp` function to source exported functions from a file
- `cppFunction` and `evalCpp` functions for inline declarations and execution
- `Rcpp::depends` attribute for specifying additional build dependencies for `sourceCpp`

Attributes can also be used for package development via the `compileAttributes` function, which generates an **Rcpp** module for all exported functions within a package.

Attributes derive their syntax from C++11 style attributes (Maurer and Wong, 2008) and are included in source files using specially formatted comments.

2 Sourcing C++ Functions

The `sourceCpp` function parses a C++ file and looks for functions marked with the `Rcpp::export` attribute. A shared library is then built and its exported functions are made available as R functions in the specified environment. For example, this source file contains an implementation of convolve (note the `Rcpp::export` attribute in the comment above the function):

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector convolveCpp(NumericVector a, NumericVector b) {

    int na = a.size(), nb = b.size();
    int nab = na + nb - 1;
    NumericVector xab(nab);

    for (int i = 0; i < na; i++)
        for (int j = 0; j < nb; j++)
            xab[i + j] += a[i] * b[j];

    return xab;
}
```

The addition of the export attribute allows us to do this from the R prompt:

```
> sourceCpp("convolve.cpp")
> convolveCpp(x, y)
```

We can now write C++ functions using standard C++ types and then source them just like an R script using the `sourceCpp` function. Any types that can be marshaled with `as` and `wrap` can be used in the signatures of exported functions (and since as and wrap are in turn extensible, a wide variety of custom types can be supported).

You can change the name of the exported function as it appears to R by adding a name parameter to `Rcpp::export`. For example the following will export `convolveCpp` as a hidden R function:

```
// [[Rcpp::export(".convolveCpp")]]
NumericVector convolveCpp(NumericVector a, NumericVector b)
```

The `sourceCpp` function performs caching based on the last modified date of the source file so as long as the source file does not change the compilation will occur only once per R session.

3 Importing Dependencies

It's also possible to use the `Rcpp::depends` attribute to declare dependencies on other packages. For example:

```
// [[Rcpp::depends(RcppArmadillo)]]

#include <RcppArmadillo.h>
using namespace Rcpp

// [[Rcpp::export]]
List fastLm(NumericVector yr, NumericMatrix Xr) {

    int n = Xr.nrow(), k = Xr.ncol();

    arma::mat X(Xr.begin(), n, k, false);
    arma::colvec y(yr.begin(), yr.size(), false);

    arma::colvec coef = arma::solve(X, y);
    arma::colvec resid = y - X*coef;

    double sig2 = arma::as_scalar(arma::trans(resid)*resid/(n-k));
    arma::colvec stderrest = arma::sqrt(
          sig2 * arma::diagvec( arma::inv(arma::trans(X)*X)) );

    return List::create(Named("coefficients") = coef,
                        Named("stderr")       = stderrest);
}
```

The inclusion of the `Rcpp::depends` attribute causes `sourceCpp` to configure the build environment to correctly compile and link against the **RcppArmadillo** package. Source files can declare more than one dependency either by using multiple `Rcpp::depends` attributes or with syntax like this:

```
// [[Rcpp::depends(Matrix, RcppArmadillo)]]
```

Dependencies are discovered both by scanning for package include directories and by invoking **inline** plugins if they are available for a package.

4 Using C++ Inline

Maintaining C++ code in it's own source file provides several benefits including the ability to use C++ aware text-editing tools and straightforward mapping of compilation errors to lines in the source file. However, it's also possible to do inline declaration and execution of C++ code. This is accomplished by either passing a code string to `sourceCpp` or using the shorter-form `cppFunction` or `evalCpp` functions. For example:

```
> cppFunction('
    int fibonacci(const int x) {
        if (x < 2)
            return x;
        else
            return (fibonacci(x - 1)) + fibonacci(x - 2);
    }
')

> evalCpp('std::numeric_limits<double>::max()')
```

You can also specify a depends parameter to `cppFunction` or `evalCpp`:

```
> cppFunction(depends = 'RcppArmadillo', code = '...')
```

Note that using `sourceCpp`, `cppFunction`, and `evalCpp` require that C++ development tools be available to build the code. If you want to distribute **Rcpp** code to users that don't have these tools installed you can bundle your code into an R package. The next section describes how you can use **Rcpp** attributes for package development.

5 Package Development

5.1 Exporting R Functions

C++ source code that uses attributes to export R functions can also be included in an R package. In this case rather than calling `sourceCpp` on individual files you call a single utility function for the whole package. The `compileAttributes` function scans the source files within a package for export attributes and generates code as required.

For example, executing this from within the package working directory:

```
> compileAttributes()
```

Results in the generation of the following two source files:

- `src/RcppExports.cpp` – An **Rcpp** module that exports the functions
- `R/RcppExports.R` – The R code required to load the **Rcpp** module

The generated code deals only with interface of functions rather than the implementation, so `compileAttributes` needs to be run only when functions are added, removed, or have their signatures changed.

5.2 Providing a C++ Interface

You can use the `Rcpp::interfaces` attribute to expose the underlying C++ functions directly to users of your package. For example, the following specifies that both R and C++ interfaces should be generated:

```
// [[Rcpp::interfaces(r, cpp)]]
```

The `Rcpp::interfaces` attribute is specified on a per-source file basis. If you request a `cpp` interface for a source file then `compileAttributes` does the following:

1. Bindings are generated into a header file located in the `inst/include` directory of the package using the naming convention *PackageName.h*
2. The generated header file enables calling the exported C++functions without any linking dependency on the package. This is based on using the `R_RegisterCCallable` and `R_GetCCallable` functions described in '*Writing R Extensions*' (R Development Core Team, 2012).
3. The exported functions are defined within a C++ namespace that matches the name of the package.

For example, an exported C++ function `bar` could be called from package `MyPackage` as follows:

```
// [[Rcpp::depends(MyPackage)]]

#include <MyPackage.h>

void foo() {
    MyPackage::bar();
}
```

Note that the default behavior if an `Rcpp::interfaces` attribute is not included in a source file is to generate an R interface only.

5.3 Using Roxygen

The **roxygen2** package (Wickham, Danenberg, and Eugster, 2011) provides a facility for automatically generating R documentation files based on specially formatted comments in R source code.

If you include roxygen comments in your C++ source file with a `//'` prefix then `compileAttributes` will transpose them into R roxygen comments within `R/RcppExports.R`. For example the following code in a C++ source file:

```
//' The length of a string (in characters).
//'
//' @param str input character vector
//' @return characters in each element of the vector
// [[Rcpp::export]]
NumericVector strLength(CharacterVector str)
```

Results in the following code in the generated R source file:

```
#' The length of a string (in characters).
#'
#' @param str input character vector
#' @return characters in each element of the vector
strLength <- function(str)
```

5.4 Packages and sourceCpp

One of the goals of **Rcpp** attributes is to simultaneously facilitate ad-hoc and interactive work with C++ while also making it very easy to migrate that work into an R package. Two major benefits of moving code from a standalone C++ source file to a package are:

1. Users without C++ development tools available can use your code.
2. Multiple source files and their dependencies are handled automatically by the R package build system.

Once you've migrated C++ code into a package it's still possible use `sourceCpp` with it for iterative development. The main thing to keep in mind is that the dependencies for source files within a package are derived from the `Depends` and `LinkingTo` fields in the package `DESCRIPTION` file rather than the `Rcpp::depends` attribute.

References

Dirk Eddelbuettel and Romain François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. URL http://www.jstatsoft.org/v40/i08/.

Dirk Eddelbuettel and Romain François. *Rcpp: Seamless R and C++ Integration*, 2012. URL http://CRAN.R-Project.org/package=Rcpp. R package version 0.9.13.

Jens Maurer and Michael Wong. Towards support for attributes in C++ (revision 6). In *JTC1/SC22/WG21 - The C++ Standards Committee*, 2008. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf. N2761=08-0271.

R Development Core Team. *Writing R extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL http://CRAN.R-Project.org/doc/manuals/R-exts.html. ISBN 3-900051-11-9.

Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel, and Romain François. *inline: Inline C, C++, Fortran function calls from R*, 2012. URL http://CRAN.R-Project.org/package=inline. R package version 0.3.10.

Hadley Wickham, Peter Danenberg, and Manuel Eugster. *roxygen2: In-source documentation for R*, 2011. URL http://CRAN.R-Project.org/package=roxygen2. R package version 2.2.12.